CS 117 Final Project Report - Dennis Ong

1. Project Overview

The main problem of this project is recreating a three-dimensional model of a teapot by using two-dimensional image scans of the teapot. These images of the teapot are taken at various angles and will be used to make multiple meshes that can be combined into one. The overall goal of this project is to create a three-dimensional model that is fairly accurate in shape and color, but smaller goals of this project include integrating proper mesh creation, pruning, smoothing, and combination techniques.

2. Data

There are two main sets of data that are used to create the three-dimensional model of the teapot - calibration images and the actual image scans of the teapot. In this project, two calibration images of chessboards were used to calibrate the positions and orientations of two cameras. For the image scans, there are 7 individual angles of the teapot. Each of these angles contains one background image and one "color" image in order to properly retrieve accurate colors of the object from the background, along with creating a proper mask. In addition to those two images , for each angle, there are 40 sets of two images (inverse of each other) with various lighting that allow a compilation of images into gray word. All the images that were previously mentioned have a resolution of 1920 pixels (width) by 1200 pixels (height).

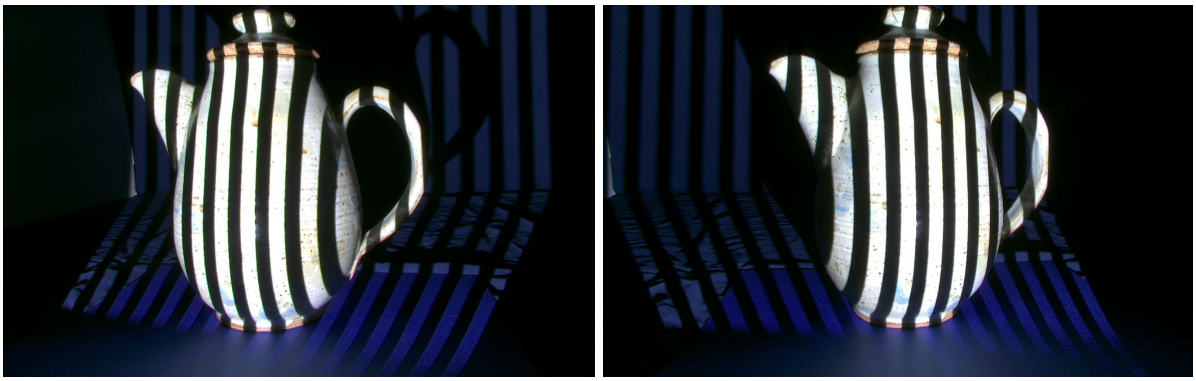


Figure 1: Example of set of two images to compile gray word

3. Algorithms

findChessboardCorners (OpenCV function) - This function was used for calibrating the extrinsic parameters of each of the cameras, left and right. This function works by taking an input of the two calibration chessboard images and number of inner corners for a chessboard row and column; it returns the estimated coordinates of the inner corners that can then be used to calibrate the cameras.

calibratePose (provided code in camutils.py) - This function performs the actual calibration of the camera by changing its rotation matrix and translation vector. It takes in a set of three-dimensional coordinates, a set of two-dimensional coordinates, estimated camera, and estimated camera parameters. In this function, the residuals are calculated between the three-dimensional and two-dimensional points. With the found residuals, a least-squares problem is created and solved, which will provide more accurate extrinsic parameters for the inputted camera. This is done for both created cameras.

reconstructNew (derived from provided code in camutils.py) - This algorithm reconstructs a three-dimensional set of points that, when plotted, should resemble the section of the object in the provided image set. This works as:

1. Decode set of left and right images (individually) to create 10-bit gray code pattern.
2. Create the left and right image masks from the gray code.
3. Calculate difference of RGB values of object and background using the distance formula in order to properly mask the teapot.
    a. Distance = $\text{sqrt}((R_{obj} - R_{bg})^2 + (G_{obj} - G_{bg})^2 + (B_{obj} - B_{bg})^2)$
4. Set left and right masks to $mask_L$ * $dist_L$ and $mask_R$ * $dist_R$, respectively
5. Find corresponding pixels between left and right images by using *numpy.intersect1d* to retrieve matching indices and find left and right coordinates of object in image.
6. Use *triangulate (provided in camutils.py)* with left and right coordinates of object in image and cameras to get triangulated three-dimensional points.
7. Use left and right coordinates of object image to find RGB values for each coordinate/pixel, then average left and right into one RGB-valued array.
8. Return left and right image coordinates, RGB values, and three-dimensional coordinates.

triangulate (provided in camutils.py) - This function triangulates a pair of coordinate sets (typically left and right cameras) and returns the triangulated three-dimensional coordinates. It begins by finding the relative pose of the two cameras with these two formulas:

1. $relRotation = (rotMatrix_{leftCam})^{-1} \text{ @ } rotMatrix_{rightCam}$
2. $relTranslation = (rotMatrix_{leftCam})^{-1} \text{ @ } (translation_{right} - translation_{left})$

To break down the triangulation into steps:

1. For each point in the left and right coordinate sets, q_L and q_R must be calculated, which are the image coordinates in each camera for each point.

2. After finding those values, the variable *A* must be created by concatenating (on axis 1) these two matrices:
    a. A_1 = q_L / $focalLength_L$
    b. A_2 = -relRotation @ (q_R / $focalLength_R$)
3. Solve for *x* in *Ax = t* with using least-squares (*numpy.linalg.lstsq)* in order to find *z*-values for both left and right cameras.
4. Calculate p_L and p_r with these formulas:
    a. p_L = q_L * (z_L / $focalLength_L$)
    b. p_R = q_R * (z_R / $focalLength_R$)
5. Transform newly created coordinates on both cameras with:
    a. p_{worldL} = $rotMatrix_L$ @ p_L + $translation_{left}$
    b. p_{worldR} = $rotMatrix_R$ @ p_R + $translation_{right}$
6. Return the averaged three-dimensional coordinates.

decode (provided code in camutils.py) - This function loads in the set of gray code images and returns the actual gray code and generated image mask. It begins by checking through each pair of images and checks if the first image is greater than or less than the second. There is a check to see if the difference is smaller than a defined threshold, which can allow excluding of specific pixels. At the end, when there are ten binary images, the function converts the 10 bit gray code to standard (base 2) binary to XOR the bits. To conclude, the binary is converted to decimal with the following formula:

1. $\sum_{n=0}^{9} B[9-n] * 2^n$, with *B* = binary created from gray code

createMesh (derived from professor-provided code on Piazza) - This algorithm prunes through points with bounding box pruning and creates a mesh through the use of Delauney triangulation. It takes in an input of left and right two-dimensional coordinates, RGB values, three-dimensional coordinates, triangle edge length threshold, and a bounding box. This works as:

1. Save points that are in designated bounding box
2. Prune out (remove) points that are not in box from inputted two-dimensional and three-dimensional coordinates and RGB values.
3. Create list of triangles with *scipy.spatial.Delaunay* function from either one of left or right two-dimensional coordinates.
4. Create each of the three edges of triangles by indexing the three-dimensional coordinates with list of triangles.
5. Check if all triangle edges are under the designated threshold.
6. Keep all triangles that have all three edges under threshold.
7. Return two-dimensional coordinates, three-dimensional coordinates, RGB values, and triangles, all have which been pruned with box bounding and triangle pruning.

4. Results

Figure 2: Individually created meshes (before alignment)

These results are from each individual meshing of respective photos. These results are from before aligning all of the meshes together to create one three-dimensional model.

5. Assessment and Evaluation

To begin, I am not very satisfied with the results. I ran into various issues through the assignment. I believe that the mesh generations, with computer algorithms were fairly simple to create, but quite difficult to perfect. As shown in my results, there are many spots with holes or incorrect colors in the mesh. A majority of the difficulty of the algorithm came from improving the meshes with certain parameters and thresholds, rather than the algorithms themselves.

One limitation of my approach was that too much data was lost in the creation of the meshes of the different parts of the teapot. As shown in the results, the holes and gaps that were left in the meshes did not allow me to properly construct a final, single three-dimensional figure. Some parts, in Meshlab, would align properly, but other parts would not align and crash Meshlab due to insufficient data. As shown in *Figure 4* below, one side of the teapot aligned quite nicely, but lacked enough data from the other parts of the teapot to finish the model.

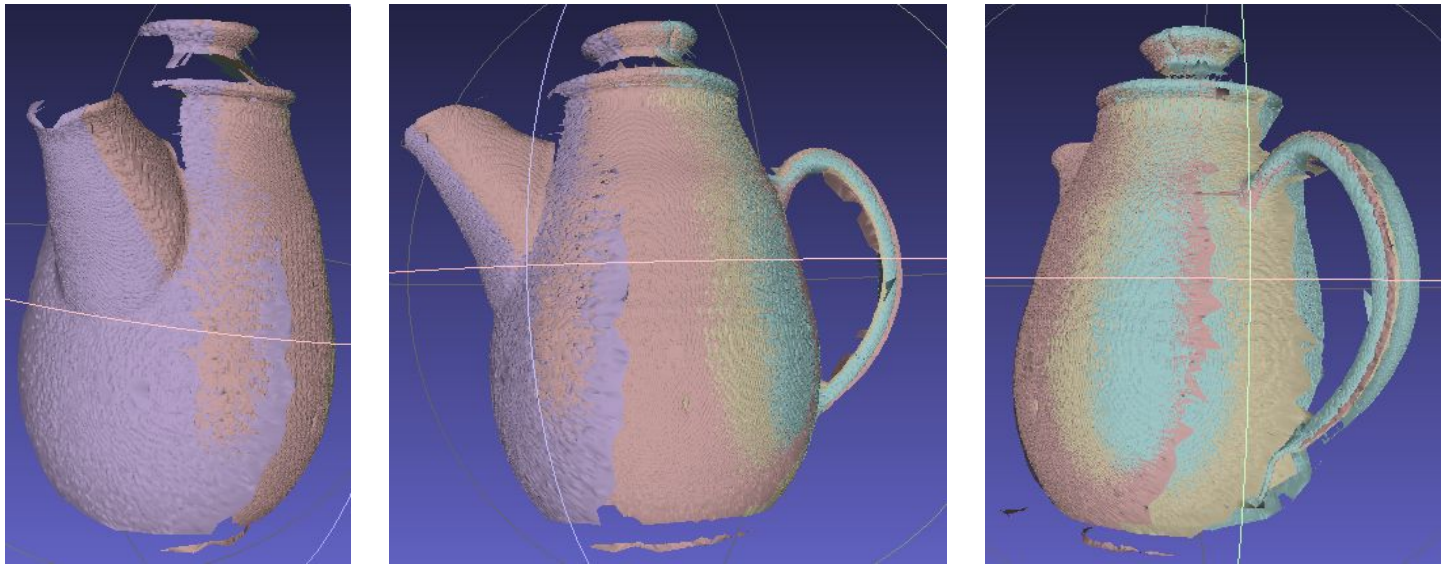


Figure 3: Front, side, and back of aligned meshes

As you can see, the entire “good” side of the teapot appears to be watertight, but the lost data at the bottom and top make it difficult to align the other meshes properly. Shown below, in *Figures 5 and 6*, the meshes seem as if they fit properly, but they lack common points to align properly.

In Meshlab, four sets of common points are recommended. Even with a guess-and-check method for points that seemed to be aligned, there was an issue with how Meshlab would abort the alignment process if there were not enough nodes with active links to each other. Because of this, I failed to build a final model.

Successfully, however, I believe the mesh cleanup went extremely well. From what I observed of the individual meshes, the bounding box pruning helped a lot in removing excess coordinates. Specifically, the bounding box helped remove the exceptionally large amount of coordinates from the shadow of the teapot. After pruning those edges, the triangle pruning also went extremely well. With the pruning, there were no excessively large triangles that formed remarkably long, jagged edges. The pruning size seemed perfect to keep the shape of the meshes.

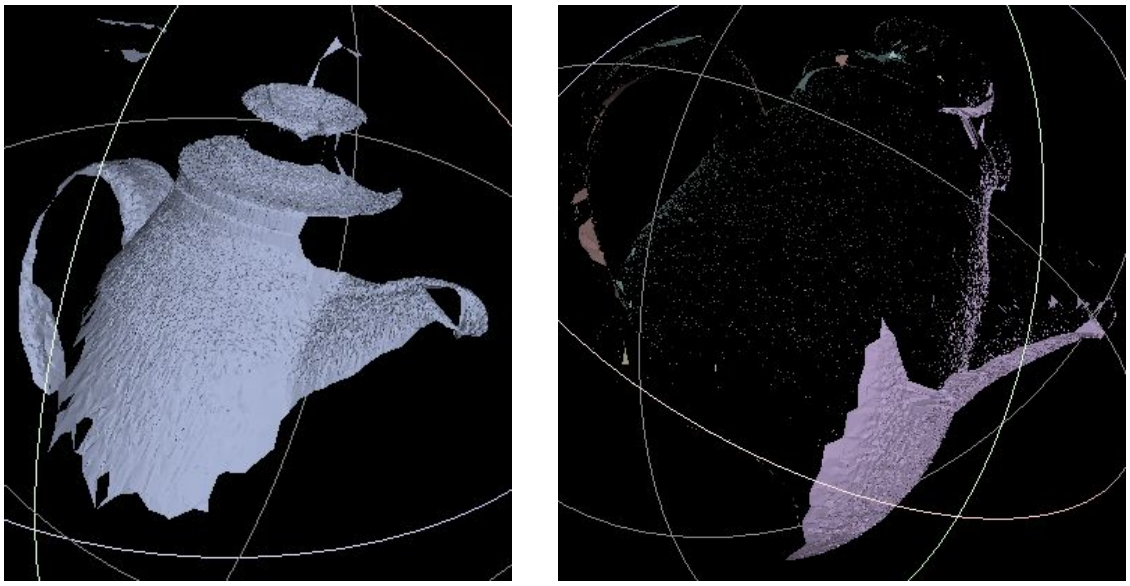
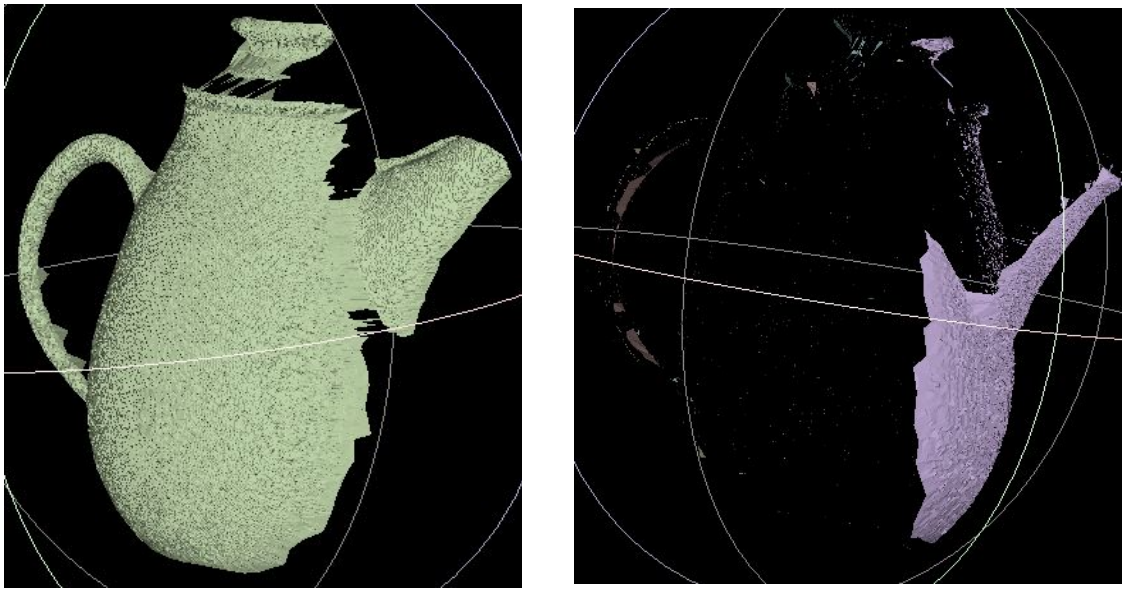


Figure 4 (above) & 5 (below): Another mesh without enough common points (Left) Single generated mesh (Right) Four aligned meshes from Figure 4



If I had more time to improve the modeling system, I believe better thresholding and masking for the photos would have been beneficial in extracting more data out of the photo sets. There was much difficulty in finding thresholds that kept a majority of the teapot intact after masking. In addition, I would attempt to create my own scans to keep the lighting more consistent across the various photos. As shown below, the mesh alignment has blotchy coloring because of the inconsistent lighting across all the photosets.

Figure 6: Bad coloring with various coloring after mesh alignment

6. Appendix

1. Calibrating extrinsic parameters
    a. Used instructor-provided *calibrate.py* to generate *.pickle* file
    b. Created parameter extraction code
2. Calculating intrinsic parameters
    a. Used instructor-provided code from assignment #3, problem 4.1, to use *findChessboardCorners* from OpenCV
    b. Created 3-d point generation code and parameter estimation values
    c. Used instructor-provided *calibratePose* function from *camutils.py*
3. Improving reconstruction function (*reconstructNew*)
    a. Modified *reconstruct* function from *camutils.py* with the following:
        i. Created object and background subtraction code for masking
        ii. Created code to store RGB values in *3xN* array
4. Created mesh function (*createMesh*)
    a. Used instructor-provided code for bounding box pruning
    b. Used instructor-provided code to create triangles by using *scipy.spatial.Delaunay* and *simplices*
    c. Created code for pruning longer triangle edges
5. Created reconstruct mesh function (*reconstructMesh*)
    a. Created code for running reconstruct and create in one function
    b. Used instructor-provided *writeply* function from *meshutils.py*
6. Created code to run *reconstructMesh* on all sets of images
    a. Created thresholding for each individual set of images (threshold for decoding (*thresh*), threshold for image mask (*threshC*), threshold for triangle edge length (*threshTri*))
    b. Created bounding boxes for each individual set of images
7. Done!